Charon Message-passing Toolkit for Scientific Computations

Rob F. Van der Wijngaart

Computer Sciences Corporation
NASA Ames Research Center, Moffett Field, CA 94035, USA
wijngaar@nas.nasa.gov

Abstract. Charon is a library, callable from C and Fortran, that aids the conversion of structured-grid legacy codes—such as those used in the numerical computation of fluid flows—into parallel, high-performance codes. Key are functions that define distributed arrays, that map between distributed and non-distributed arrays, and that allow easy specification of common communications on structured grids. The library is based on the widely accepted MPI message passing standard. We present an overview of the functionality of Charon, and some representative results.

1 Introduction

A sign of the maturing of the field of parallel computing is the emergence of facilities that shield the programmer from low-level constructs such as message passing (MPI, PVM) and shared memory parallelization directives (P-Threads, OpenMP, etc.), and from parallel programming languages (High Performance Fortran, Split C, Linda, etc.). Such facilities include: 1) (semi-)automatic tools for parallelization of legacy codes (e.g. CAPTools [8], ADAPT [5], CAPO [9], SUIF compiler [6], SMS preprocessor [7], etc.), and 2) application libraries for the construction of parallel programs from scratch (KeLP [4], OVERTURE [3], PETSc [2], Global Arrays [10], etc.).

The Charon library described here offers an alternative to the above two approaches, namely a mechanism for *incremental* conversion of legacy codes into high-performance, scalable message-passing programs. It does so without the need to resort up front to explicit parallel programming constructs. Charon is aimed at applications that involve structured discretization grids used for the solution of scientific computing problems. Specifically, it is designed to help parallelize algorithms that are not naturally data parallel—i.e., that contain complex data dependencies—which include almost all advanced flow solver methods in use at NASA Ames Research Center. While Charon provides strictly a set of user-callable functions (C and Fortran), it can nonetheless be used to convert serial legacy codes into highly-tuned parallel applications. The crux of the library is that it enables the programmer to codify information about existing multi-dimensional arrays in legacy codes and map between these non-distributed arrays and newly defined, truly distributed arrays *at runtime*. This allows the

programmer to keep most of the serial code unchanged and only use distributed arrays in that part of the code of prime interest. This code section, which is parallelized using more functions from the Charon library, is gradually expanded, until the entire code is converted. The major benefit of incremental parallelization is that it is easy to ascertain consistency with the serial code. In addition, the user keeps careful control over data transfer between processes, which is important on high-latency distributed-memory machines[1].

The usual steps that a programmer takes when parallelizing a code using Charon are as follows. First, define a distribution of the arrays in the program, based on a division of the grid(s) among all processors. Second, select a section of the code to be parallelized, and construct a so-called parallel bypass: map from the non-distributed (legacy code) array to the distributed array upon entry of the section, and back to the non-distributed array upon leaving it. Third, do the actual parallelization work for the section, using more Charon functions.

The remainder of this paper is structured as follows. In Section 2 we explain the library functions used to define and manipulate distributed arrays (*distributions*), including those that allow the mapping between non-distributed and distributed arrays. In Section 3 we describe the functions that can be used actually to parallelize an existing piece of code. Some examples of the use and performance of Charon are presented in Section 4.

2 Distributed Arrays

Parallelizing scientific computations using Charon is based on domain decomposition. One or more multi-dimensional grids are defined, and arrays—representing computational work—are associated with these grids. The grids are divided into nonoverlapping pieces, which are assigned to the processors in the computation. The associated arrays are thus distributed as well. This process takes place in several steps, illustrated in Fig. 1, and described below.

First (Fig. 1a), the logically rectangular discretization *grid* of a certain dimensionality and extent is defined, using `CHN_Create_grid`. This step establishes a geometric framework for all arrays associated with the grid. It also attaches to the grid an MPI [11] communicator, which serves as the context and processor subspace within which all subsequent Charon-orchestrated communications take place. Multiple coincident or non-coincident communicators may be used within one program, allowing the programmer to assign the same or different (sets of) processors to different grids in a multiple-grid computation.

Second (Fig. 1b), tessellations of the domain (*sections*) are defined, based on the grid variable. The associated library call is `CHN_Create_section`. Sections contain a number of cutting planes (*cuts*) along each coordinate direction. The grid is thus carved into a number of *cells*, each of which contains a logically rectangular block of grid points. Whereas the programmer can specify any number of cuts and cut locations, a single call to a high-level routine

[1] We will henceforth speak of *processors*, even if *processes* is the more accurate term.

often suffices to define all the cuts belonging to a particular domain decomposition. For example, defining a section with just a single cell (i.e. a non-divided grid with zero cuts) is accomplished with `CHN_Set_solopartition_cuts`. Using `CHN_Set_unipartition_cuts` divides the grid evenly into as many cells as there are processors in the communicator—nine in this case.

Third (Fig. 1c), cells are assigned to processors, resulting in a *decomposition*. The associated function is `CHN_Create_decomposition`. The reason why the creation of section and decomposition are separated is to provide flexibility. For example, we may divide a grid into ten slices for execution on a parallel computer, but assign all slices to the same processor for the purpose of debugging on a serial machine. As with the creation of sections, we can choose to assign each cell to a processor individually, or make a single call to a high-level routine. For example, `CHN_Set_unipartition_owners` assigns each cell in the unipartition section to a different processor. But regardless of the number of processors in the communicator, `CHN_Set_solopartition_owners` assigns all cells to the same processor.

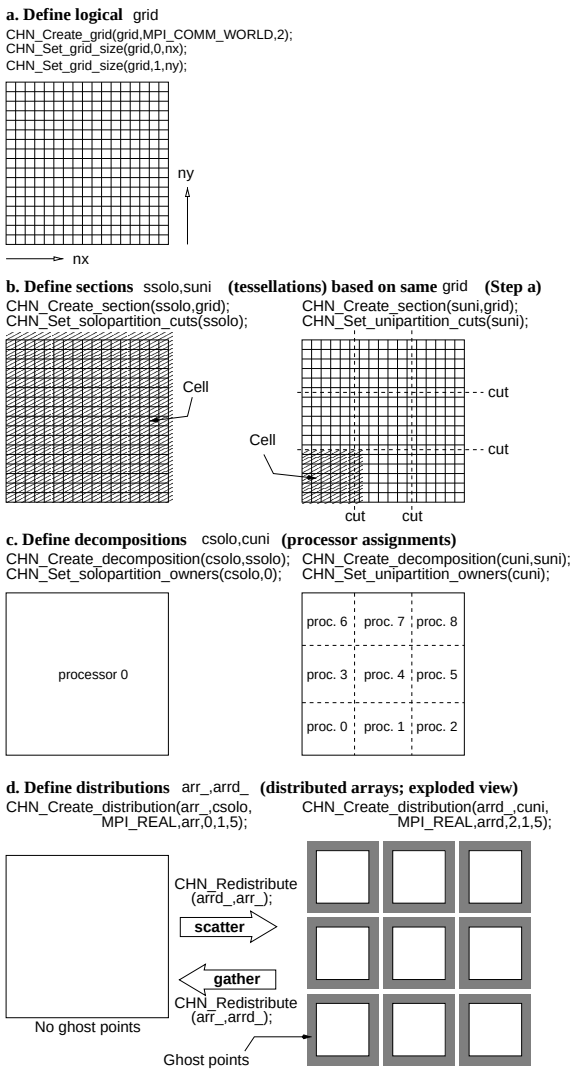

Fig. 1. Defining and mapping distributed arrays

Finally (Fig. 1d), arrays with one or more spatial dimensions (same as the grid) are associated with a decomposition, resulting in *distributions*. The associated function is `CHN_Create_distribution`. The arrays may represent scalar quantities at each grid point, or higher-order tensors. In the example in Fig. 1d the tensor rank is 1, and the thusly defined vector has 5 components at each grid point. A distribution has one of a subset of the regular

MPI data types (`MPI_REAL` in this case). Since Charon expressly supports stencil operations on multi-dimensional grids, we also specify a number of *ghost points*. These form a border of points (shaded area) around each cell, which acts as a cache for data copied from adjacent cells. In this case the undivided distribution has zero ghost points, whereas the unipartition distribution has two, which can support higher-order difference stencils (e.g. a 13-point 3D star).

The aspect of distributions that sets Charon apart most from other parallelization libraries is the fact that the programmer also supplies the memory occupied by the distributed array; Charon provides a *structuring interpretation* of user space. In the example in Fig. 1d it is assumed that `arr` is the starting address of the non-distributed array used in the legacy code, whereas `arrd` is the starting address of a newly declared array that will hold data related to the unipartition distribution. By mapping between `arr` and `arrd`—using `CHN_Redistribute` (see also Fig. 1d)—we can dynamically switch from the serial legacy code to truly distributed code, and back. All that is required is that the programmer define distribution `arr_` such that the memory layout of the (legacy code) array `arr` coincide exactly with the Charon specified layout. This act of reverse engineering is supported by functions that allow the programmer to specify array padding and offsets, by a complete set of query functions, and by Charon's unambiguously defined memory layout model (see Section 3.3).

`CHN_Redistribute` can be used not only to construct parallel 'bypasses' of serial code of the kind demonstrated above, but also to map between *any* two compatible distributions (same grid, data type, and tensor rank). This is shown in Fig. 2, where two stripwise distributions, one aligned with the first coordinate axis, and the other with the second, are mapped into each other, thereby establishing a dynamic transposition. This is useful when there are very strong but mutually incompatible data dependencies in different parts of the code (e.g. 2D FFT). By default, the unipartition decomposition divides all coordinate directions evenly, but by excluding certain directions from partitioning (`CHN_Exclude_partition_direction`) we can force a stripwise distribution.

3 Distributed and Parallel Execution Support

While it is an advantage to be able to keep most of a legacy code unchanged and focus on a small part at a time for parallelization, it is often still nontrivial to arrive at good parallel code for complicated numerical algorithms. Charon offers support for this process at two levels.

The first concerns a set of wrapping functions that allows us to keep the serial logic and structure of the legacy code unchanged, although the data is truly distributed. These functions incur a significant overhead, and are meant to be removed in the final version of the code. They provide a stepping stone in the parallelization, and may be skipped by the more intrepid programmer.

The second is a set of versatile, highly optimized bulk communication functions that support the implementation of sophisticated data-parallel and—more importantly—non-data-parallel numerical methods, such as pipelined algorithms.

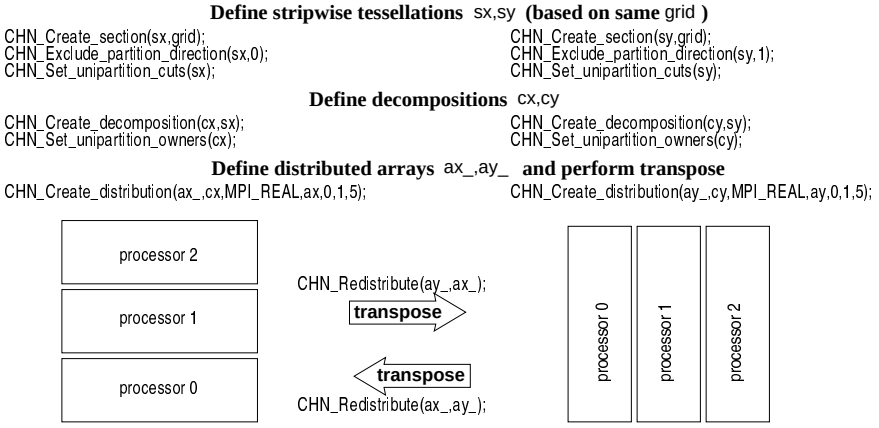

Fig. 2. Transposing distributed arrays

3.1 Wrapping Functions

In a serial program it is obvious what the statement `a(i,j,k) = b(i+2,j-1,k)` means, provided `a` and `b` have been properly dimensioned, but when these arrays are distributed across several processors the result is probably not what is expected, and most likely wrong. This is due to one of the fundamental complexities of message-passing, namely that the programmer is responsible for defining explicitly and managing the data distribution. Charon can relieve this burden by allowing us to write the above assignment as:

```
call CHN_Assign(CHN_Address(a_,i,j,k),CHN_Value(b_,i+2,j-1,k))
```

with no regard for how the data is distributed (assuming that `a_` and `b_` are distributions related to arrays `a` and `b`, respectively). The benefit of this wrapping is that the user need not worry (yet) about communications, which are implicitly invoked by Charon, as needed.

The three functions introduced here have the following properties. `CHN_Value` inspects the distribution `b_`, determines which (unique) processor owns the grid point that holds the value, and broadcasts that value to all processors in the communicator ('*owner serves*' rule). `CHN_Address` inspects the distribution `a_` and determines which processor owns the grid point that holds the value. If the calling processor is the owner, the actual address—an *lvalue*—is returned, and `NULL` otherwise[2]. `CHN_Assign` stores the value of its second argument at the address in the first argument **if** the address is not `NULL`. Consequently, only the point owner of the left hand side of the assignment stores the value ('*owner assigns*' rule). No distinction is made between values obtained through `CHN_Value` and local values, or expressions containing combinations of each; all are *rvalues*. Similarly, no distinction is made between local addresses and those obtained

[2] In Fortran return values cannot be used as lvalues, but this problem is easily circumvented, since the address is immediately passed to a C function.

through CHN_Address. Hence, the following assignments are all legitimate:

```
call CHN_Assign(CHN_Address(a_,i,j,k),5.0)                    !1
call CHN_Assign(aux,CHN_Value(b_,i,j-1,k)+1.0)                !2
aux = CHN_Value(b_,i,j-1,k)+1.0                               !3
```

It should be observed that assignments 2 and 3 are equivalent. An important feature of wrapped code is that it is completely serialized. All processors execute the same statements, and whenever an element of a distributed array occurs on the right hand side of an assignment, it is broadcast. As a result, it is guaranteed to have the correct serial logic of the legacy code.

3.2 Bulk Communications

The performance of wrapped code can be improved by removing the need for the very fine-grained, implicitly invoked communications, and replacing them with explicitly invoked bulk communications. Within structured-grid applications the need for non-local data is often limited to (spatially) nearest-neighbor communication; stencil operations can usually be carried out without any communication, provided a border of ghost points (see Fig. 1d) is filled with array values from neighboring cells. This fill operation is provided by CHN_Copyfaces, which lets the programmer specify exactly which ghost points to update. The function takes the following arguments:

- the thickness of layer of ghost points to be copied; this can be at most the number of ghost points specified in the definition of the *distribution*,
- the components of the tensor to be copied. For example, the user may wish only to transfer the diagonal elements of a matrix,
- the coordinate direction in which the copying takes place,
- the sequence number of the cut (defined in the *section*) across which copying takes place,
- the rectangular subset of points within the cut to be copied.

In general, all processors within the grid's MPI communicator execute CHN_Copyfaces, but those that do not own points involved in the operation may safely skip the call. A useful variation is CHN_Copyfaces_all, which fills all ghost points of the distribution in all coordinate directions. It is the variation most commonly encountered in other parallelization packages for structured-grid applications, since it conveniently supports data parallel computations. But it is not sufficient to implement, for example, the pipeline algorithm of Section 4.2.

The remaining two bulk communications provided by Charon are the previously described CHN_Redistribute, and CHN_Gettile. The latter copies a subset of a distributed array—which may be owned by several processors—into the local memory of a specified processor (cf. Global Arrays' ga_get [10]). This is useful for applications that have non-nearest-neighbor remote data dependencies, such as non-local boundary conditions for flow problems.

3.3 Parallelizing Distributed Code Segments

Once the remote-data demand has been satisfied through the bulk copying of ghost point values, the programmer can instruct Charon to suppress broadcasts by declaring a section of the code *local* (see below). Within a local section not all code should be executed anymore by all processors, since assignments to points not owned by the calling processor will usually require remote data not present on that calling processor; the code must be restructured to restrict the index sets of loops over (parts of) the grid. This is the actual process of parallelization, and it is left to the programmer. It is often conceptually simple for structured-grid codes, but the bookkeeping matters of changing all data structures at once have traditionally hampered such parallelization. The advantage of using Charon is that the restructuring focuses on small segments of the code at any one time, and that the starting point is code that already executes correctly on distributed data sets. The parallelization of a loop nest typically involves the following steps.

1. Determine the order in which grid cells should be visited to resolve all data dependencies in the target parallel code. For example, during the x-sweep in the SP code described in Section 4.1 (Fig. 3), cells are visited layer by layer, marching in the positive x-direction. In this step all processors still visit all cells, and no explicit communications are required, thanks to the wrapper functions. This step is supported by query functions that return the number of cells in a particular coordinate direction, and also the starting and ending grid indices of the cells (useful for computing loop bounds).
2. Fill ghost point data in advance. If the loop is completely data parallel, a single call to `CHN_Copyfaces` or `CHN_Copyfaces_all` before entering the loop is usually sufficient to fill ghost point values. If a non-trivial data dependence exists, then multiple calls to `CHN_Copyfaces` are usually required. For example, in the x-sweep in the SP code `CHN_Copyfaces` is called between each layer of cells. At this stage all processors still execute all statements in the loop nest, so that they can participate in broadcasts of data not resident on the calling processor. However, whenever it is a ghost point value that is required, it is served by the processor that owns it, rather than the processor that owns the cell that has that point as an interior point. This seeming ambiguity is resolved by placing calls to `CHN_Begin_ghost_access` and `CHN_End_ghost_access` around the code that accesses ghost point data, which specify the index of the cell whose ghost points should be used.
3. Suppress broadcasts. This is accomplished by using the bracketing construct `CHN_Begin_local/CHN_End_local` to enclose the code that accesses elements of distributed arrays. For example:

   ```
   call CHN_Begin_local(MPI_COMM_WORLD)
   call CHN_Assign(CHN_Address(a_,i),CHN_Value(b_,i+1)-CHN_Value(b_,i-1))
   call CHN_End_local(MPI_COMM_WORLD)
   ```

   At the same time, the programmer restricts accessing lvalues to points actually owned by the calling processor. This is supported by the query functions `CHN_Point_owner` and `CHN_Cell_owner`, which return the MPI rank of the processor that owns the point and the grid cell, respectively.

Once the code segment is fully parallelized, the programmer can strips the wrappers to obtain the final, high-performance code. Stripping effectively consists of translating global grid coordinates into local array indices, a chore that is again easily accomplished, due to Charon's transparent memory layout model. By default, all subarrays of the distribution associated with individual cells of the grid are dimensioned identically, and these dimensions can be computed in advance, or obtained through query functions. For example, assume that the number of cells owned by each processor is `nmax`, the dimensions of the largest cell in the grid are `nx`×`ny`, and the number of ghost points is `gp`. Then the array `w` related to the scalar distribution `w_` can be dimensioned as follows:

```
dimension w(1-gp:nx+gp,1-gp:ny+gp,nmax)
```

Assume further that the programmer has filled the arrays `beg(2,nmax)` and `end(2,nmax)` with the beginning and ending point indices, respectively (using Charon query functions), of the cells in the grid owned by the calling processor. Then the following two loop nests are equivalent, provided `n` $\leq$ `nmax`.

```
do j=beg(2,n),end(2,n)
 do i=beg(1,n),end(1,n)
  call CHN_Assign(CHN_Address(w_,i,j),5.0)
 end do
end do
```

```
do j=1,end(2,n)-beg(2,n)+1
 do i=1,end(1,n)-beg(1,n)+1
  w(i,j,n) = 5.0
 end do
end do
```

The above example illustrates the fact that Charon minimizes encapsulation; it is always possible to access data related to distributed arrays directly, without having to copy data or call access functions. This is a programming convenience, as well as a performance gain. Programs parallelized using Charon usually ultimately only contain library calls that create and query distributions, and that perform high-level communications.

Finally, it should be noted that it is not necessary first to wrap legacy code to take advantage of Charon's bulk communication facilities for the construction of parallel bypasses. Wrappers and bulk communications are completely independent.

4 Examples

We present two examples of numerical problems, SP and LU, with complicated data dependencies. Both are taken from the NAS Parallel Benchmarks (NPB) [1], of which hand-coded MPI versions (NPB-MPI) and serial versions are freely available. They have the form: $Au^{n+1} = b(u^n)$, where u is the time-dependent solution, n is the number of the time step, and b is a nonlinear 13-point-star stencil operator. The difference is in the shape of A, the discretization matrix that defines the 'implicitness' of the numerical scheme. For SP it is effectively: $A_{SP} = L_z L_y L_x$, and for LU: $A_{LU} = L_+ L_-$.

4.1 SP Code

L_z, L_y and L_x are fourth-order difference operators that determine data dependencies in the z, y, and x directions, respectively. A_{SP} is numerically inverted

in three corresponding phases, each involving the solution of a large number of independent banded (penta-diagonal) matrix equations, three for each grid line. Each equation is solved using Gaussian elimination, implemented as two sweeps along the grid line, one in the positive (forward elimination), and one in the negative direction (backsubstitution). The method chosen in NPB-MPI is the *multipartition* (MP) decomposition strategy. It assigns to each processor multiple cells such that, regardless of sweep direction, each processor has work to do during each stage of the solution process. An example of a 9-processor 3D MP is shown in Fig. 3. Details can be found in [1]. While most packages we have studied do not allow the definition of MP, it is easily specified in Charon:

```
call CHN_Create_section(multi_sec,grid)
call CHN_Set_multipartition_cuts(multi_sec)
call CHN_Create_decomposition(multi_cmp,multi_sec)
call CHN_Set_multipartition_owners(multi_cmp)
```

The solution process in the x direction is as follows (Fig. 3). All processors start the forward elimination on the left side of the grid. When the boundary of the first layer of cells is reached, the elements of the penta-diagonal matrix that need to be passed to the next layer of cells are copied in bulk using `CHN_Copyfaces`. Then the next layer of cells is traversed, followed by another copy operation, etc. The number of floating point operations and words communicated in the Charon version of the code is exactly the same as in NPB-MPI. The only difference is that Charon copies the values into ghost points, whereas in NPB-MPI they are directly used to update the matrix system without going to main memory. The latter is more efficient, but comes at the cost of a much greater program complexity, since the communication must be fully integrated with the computation.

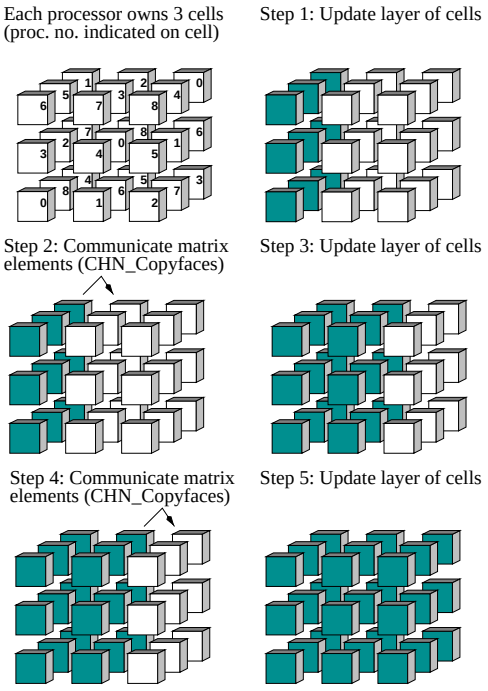

Fig. 3. Nine-processor multipartition decomposition, and solution process for SP code (L_x: forward elimination)

The results of running both Charon and NPB-MPI versions of SP for three different grid sizes on an SGI Origin2000 (250 MHz MIPS R10000) are shown in Fig. 4. Save for a deterioration at 81 processors for class B (102^3 grid) due to a bad stride, the results indicate that the Charon version achieves approximately 70% of the performance of NPB-MPI, with roughly the same scalability charac-

teristics. While 30% difference is significant, it should be noted that the Charon version was derived from the serial code in three days, whereas NPB-MPI took more than one month (both by the same author). Moreover, since Charon and MPI calls can be freely mixed, it is always possible for the programmer who is not satisfied with the performance of Charon communications to do (some of) the message passing by hand.

If SP is run on a 10-CPU Sun Ultra Enterprise 4000 Server (250 MHz Ultrasparc II processor), whose cache structure differs significantly from the R10000, the difference between results for the Charon and NPB-MPI versions shrinks to a mere 3.5%; on this machine there is no gain in performance through hand coding.

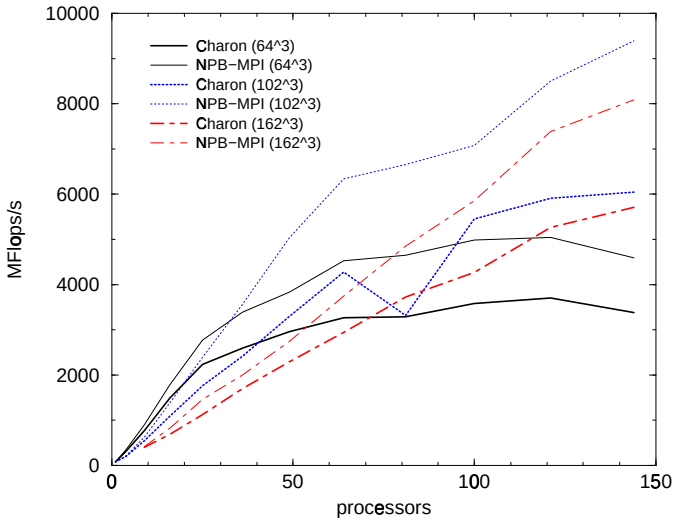

Fig. 4. Performance of SP benchmark on 250 MHz SGI Origin2000

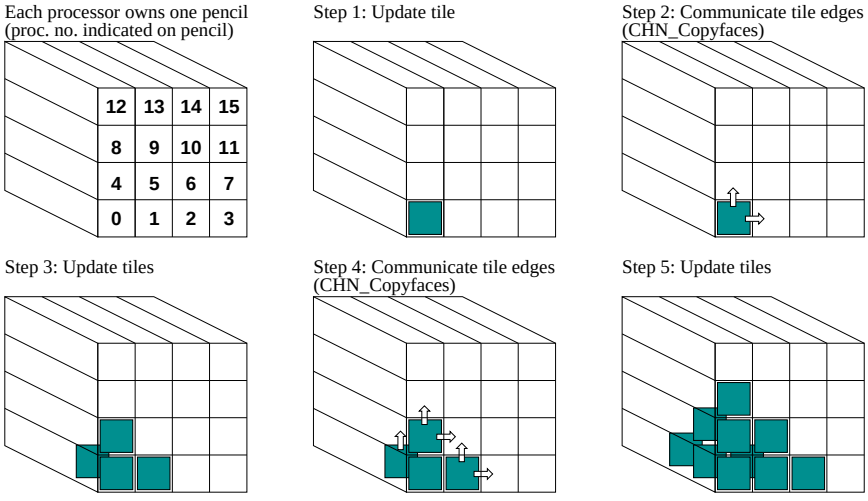

Fig. 5. Unipartition pencil decomposition (16 processors); Start of pipelined solution process for LU code (L_-)

4.2 LU Code

L_- and L_+ are first-order direction-biased difference operators. They define two sweeps over the entire grid. The structure of L_- dictates that no point (i,j,k) can be updated before updating all points (i_p,j_p,k_p) with smaller indices: $\{(i_p,j_p,k_p)|i_p \leq i, j_p \leq j, k_p \leq k, (i_p,j_p,k_p) \neq (i,j,k)\}$. This data dependency is the same as for the Gauss-Seidel method with lexicographical point ordering. L^+ sweeps in the other direction. Unlike for SP, there is no concept of independent grid lines for LU. The solution method chosen for NPB-MPI is to divide the grid into pencils, one for each processor, and pipeline the solution process, Fig. 5;

```
call CHN_Create_section(pencil_sec,grid)
call CHN_Exclude_partition_direction(pencil_sec,2)
call CHN_Set_unipartition_cuts(pencil_sec)
call CHN_Create_decomposition(uni_cmp,uni_sec)
call CHN_Set_unipartition_owners(uni_cmp)
```

Each unit of computation is a single plane of points (*tile*) of the pencil. Once a tile is updated, the values on its boundary are communicated to the pencil's Eastern and Northern (for L_-) neighbors. Subsequently, the next tile in the pencil is updated. Of course, not all boundary points of the whole pencil should be transferred after completion of each tile update, but only those of the 'active' tile. This is easily specified in `CHN_Copyfaces`.

The results of both Charon and NPB-MPI versions of LU for three different grid sizes on the SGI Origin are shown in Fig. 6. Now the performance of the Charon code is almost the same as that of NPB-MPI. This is because both use ghost points for transferring information between neighboring pencils. Again, on the Sun E4000, the performances of the hand coded and Charon parallelized programs are nearly indistinguishable.

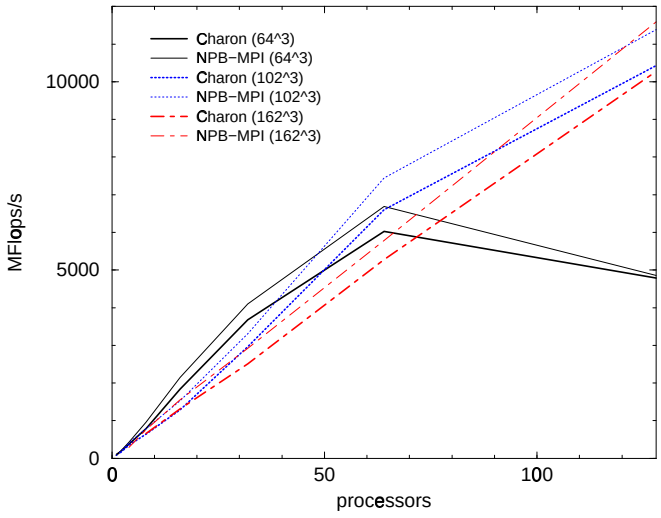

Fig. 6. Performance of LU benchmark on 250 MHz SGI Origin2000

5 Discussion and Conclusions

We have given a brief presentation of some of the major capabilities of Charon, a parallelization library for scientific computing problems. It is useful for those ap-

plications that need high scalability, or that have complicated data dependencies that are hard to resolve by analysis engines. Since some hand coding is required, it is more labor intensive than using a parallelizing compiler or code transformation tool. Moreover, the programmer must have some knowledge about the application program structure in order to make effective use of Charon. When the programmer does decide to make the investment to use the library, the results are close to the performance of hand-coded, highly tuned message passing implementations, at a fraction of the development cost. More information on the toolkit and a user guide are being made available by the author [12].

References

1. D.H. Bailey, T. Harris, W.C. Saphir, R.F. Van der Wijngaart, A.C. Woo, M. Yarrow, "The NAS parallel benchmarks 2.0," Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, December 1995.
2. S. Balay, W.D. Gropp, L. Curfman McInnes, B.F. Smith, "PETSc 2.0 Users manual," Report ANL–95/11 - Revision 2.0.24, Argonne National Laboratory, Argonne, IL, 1999.
3. D.L. Brown, G.S. Chesshire, W.D. Henshaw, D.J. Quinlan, "Overture: An object-oriented software system for solving partial differential equations in serial and parallel environments," 8^{th} SIAM Conf. Parallel Proc. for Scientific Computing, Minneapolis, MN, March 1997.
4. S.B. Baden, D. Shalit, R.B. Frost, "KeLP User Guid Version 1.3," Dept. Comp. Sci. and Engin., UC San Diego, La Jolla, CA, January 2000.
5. M. Frumkin, J. Yan, "Automatic Data Distribution for CFD Applications on Structured Grids," NAS Technical Report NAS-99-012, NASA Ames Research Center, CA, 1999.
6. M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S.-W. Liao, E. Bugnion, M.S. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler," IEEE Computer, Vol. 29, pp. 84–89, December 1996.
7. T. Henderson, D. Schaffer, M. Govett, L. Hart, "SMS Users Guide," NOAA/Forecast Systems Laboratory, Boulder, CO, January 2000.
8. C.S. Ierotheou, S.P. Johnson, M. Cross, P.F. Leggett, "Computer aided parallelisation tools (CAPTools)—conceptual overview and performance on the parallelisation of structured mesh codes," Parallel Computing, Vol. 22, pp. 163–195, 1996.
9. H. Jin, M. Frumkin, J. Yan, "Use Computer-aided tools to parallelize large CFD applications," NASA High Performance Computing and Communications Computational Aerosciences (CAS) Workshop 2000, NASA Ames Research Center, Moffett Field, CA, February 2000.
10. J. Nieplocha, R.J. Harrison, R.J. Littlefield, "The global array programming model for high performance scientific computing," SIAM News, Vol. 28, August-September 1995.
11. M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, J. Dongarra, "MPI: The Complete Reference," MIT Press, 1995.
12. R.F. Van der Wijngaart, Charon home page, http://www.nas.nasa.gov/~wijngaar/charon.